Notes for the DOS port of Microwindows/Nano-X

Nano-X is primarily intended to be used with Linux. So it comes as source code which shall be compiled on the target system using the gcc compiler.

DJGPP is a DOS port of the gcc compiler which comes with a lot of Linux GNU utilities ported to DOS, such as the Make program. So I used that for the port of Nano-X to DOS.

These notes describe how to port the original Linux code to DOS. If you downloaded my source code package from the google.code site, these changes have already been applied.

Nano-X comes with two different APIs: one that is compatible to the Microsoft Windows-API and one that is compatible to the X-Windows API. So after compiling Nano-X you will have two libraries you can link with your application program: libmwin.a if you want to use the Windows-API and libnano-X.a if you want to use the X-Windows API.

1. Setup DJGPP

If you have not installed DJGPP yet, you should download the following files, if you have DJGPP already make sure you have them installed:

First enter the address below into your browsers address field or just click on this link:

ftp://ftp.delorie.com/pub/djgpp/current/

In this FTP directory there are several subdirectories. You should download the following files:

```
V2: readme.1st, djtzn203.zip, djdev204.zip(of the beta directory!)
V2gnu: gcc453b.zip, bnu281b.zip, mak3791b.zip
V2misc: pmode13b.zip
```

To display JPEG, TIFF and PNG images you will need:

```
V2tk: jpeg8bb.zip, png152b.zip, zlib125b, tiff395b.zip
```

I recommend to install these files too which are required to port other Linux programs:

```
V2gnu: gpp453b.zip, acnf257b.zip, amak175b.zip, bsh203b.zip,
dif30b,zip, fil41b,zip, find41b.zip, grep28b.zip, pat261b.zip,
perl552.b.zip, sed421b.zip, shl2011b.zip, txt20b.zip,
pdcur34a.zip, txi40b.zip,
```

Unzip these files so you get a c:\djgpp directory which contains all these in the appropiate subdirectories.

2. Setup the DJGPP environment

To compile Nano-X you will need long filename support for DOS. So I compiled it in a DOS window of Windows XP. If you compile in real mode DOS, you have to load DOSLFN to get long filename support. However, compilation takes much longer than in a DOS window of Windows XP since Windows has a large disk buffer which helps a lot with these large files.

Before using DJGPP I run the following batch file to setup the environment as necessary:

```
rem Put the djgpp path before the DOS command path or GNU find will not work
path=c:\usr\local\bin;c:\usr\local;c:\djgpp\bin;c:\djgpp\mwin\src\lib;%path%
set DJDIR=c:\djgpp
rem Sets e.g. the include directory path
set DJGPP=c:\djgpp\djgpp.env
rem need to set this to get truetype fonts to work in nano-x
set TTFONTDIR=c:\usr\share\fonts\truetype\ttf-dejavu
rem set TTFONTDIR=c:\djgpp\mwin\src\fonts\truetype
rem set FLTK_SCHEME=plastic
rem set FLTK_SCHEME=gtk+
set FLTK_SCHEME=grad1
```

I include a hello.c source code so you can test if the djgpp installation works.

<u>3. Install and compile the Nano-X sources</u>

After downloading the latest version of Nano-X, use 7zip to unpack it into a directory e.g.:

c:\djgpp\microwin

In the microwin directory there is a src directory which is the base directory from where you have to run the Make utility to compile Nano-X. So if you are in the command line window you go into that directory and enter Make on the command line.

The Nano-X package currently has several depreciated makefiles and config settings for DOS which will not work. Therefore I wrote new Makefiles which can be used with DJGPP.

Before running Make you have to install the these Makefiles I have written for DJGPP in their respective directories:

```
Makefile -> c:\djgpp\microwin\src
Makefile-mwin -> c:\djgpp\microwin\src\mwin (rename Makefile-mwin to Makefile)
Makefile-nanox -> c:\djgpp\microwin\src\nanox (rename Makefile-nanox to Makefile)
Makefile-bmp -> c:\djgpp\microwin\src\mwin\bmp (rename Makefile-bmp to Makefile)
Makefile-engine -> c:\djgpp\microwin\src\engine (rename Makefile-engine to Makefile)
Makefile-fonts -> c:\djgpp\microwin\src\fonts (rename Makefile-fonts to Makefile)
Makefile-drivers -> c:\djgpp\microwin\src\drivers (rename Makefile-drivers to Makefile)
```

There is a makecopy.bat file in the djgpp directory which shall do these copy operations for you.

The Makefile you have copied into c:\djgpp\microwin\src is the main Makefile you will run when you start Make from the command line in that directory. This file has several configuration options you can set.

Before running make make sure the three drivers mentioned in section four below are installed in the "c:\djgpp\microwin\src\drivers" directory. Also copy the X6x13.c font into the fonts directory.

This Makefile will first build the libmwin.a, the libnano-X.a and the libimages.a libraries. For this it calls the different Makefiles you copied into the subdirectories. The new libraries will be in the "lib" subdirectory when done.

Then the Makefile will compile the demos for the Windows-API and the XWindows-API. These will be in the "bin" directory as exe files where you can run them.

When the screen driver set the display in SVGA mode, it also redirects all STDOUT output to the NUL device, i.e. discards it. Otherwise printf() commands in the code will corrupt the display.

If you want to get the output from print statements in the code you can run the programs with the "redir" command to send stderr and stdout to a file. I include a fr.bat file for that.

If you ran the program in a Windows DOS box and entered Alt-Enter to get to the Windows desktop, the screen colors will be corrupted when you return to the program. This is because Windows will not restore the SVGA mode. You can enter ALT-Comma then to go to text mode and ALT-Comma again will restore the screen properly.

4. Screen drivers for DOS

The screen driver interface did change considerably between version 0.92 and 0.93. So I wrote two new DOS screen drivers for version 0.93.

One is based on the GRX graphics library like the DOS screen driver in the previous versions.

Since only a very tiny fraction of the GRX library's functionality is used by that screen driver I wrote a second one which uses the VESA interface of the video graphics cards directly. This driver needs less memory space since it does not load the GRX library and seems to work a little bit faster too.

The driver using the GRX library is called scr_djgrx.c and the other one scr_djvesa.c / djvesa.h. Both have to be in the c:\djgpp\microwin\src\drivers directory.

These drivers work with 16bit, 24bit and 32bit TrueColor settings. I could not test the 8bit palette setting.

You have to set the desired screen resolution and pixel format in the main Makefile before compiling Nano-X. If you change the settings, do a "Make clean" before the next Make, since several modules need to read the new settings and adjust themselves accordingly.

I also wrote a modified keyboard driver for DOS since the interface for that also changed with version 0.93. Its name is "kbd_dj.c". The driver uses BIOS and DOS calls to retrieve the scancode as well as the keyvalue. This approach uses the international keyboard support of the DOS keyboard driver and there is no need for several different drivers in Nano-X. It also simulates make and break codes and includes a conversion table from IBM codepage 850 to iso8859-1.

Finally I wrote a new mouse driver, "mou_dj.c". This is event driven and registers an interrupt with the DOS mouse driver.

There is an environment variable which Nano-X reads and which allows to set the screen resolution and pixel type. This variable is called "NANOSCR" which is defined as:

NANOSCR=[SCREEN_WIDTH] [SCREEN_HEIGHT] [SCREEN_PIXTYPE]

For SCREEN_PIXTYPE the following values are valid: 8888 for 32bit, 888 for 24bit, 565 for 16bit and 8 for 8bit. An example would be:
"set NANOSCR=800 600 565" for 800x660 and TRUECOLOR565

<u>5. Images</u>

Nano-X supports the display of BMP, GIF, PNM and XPM images directly. For other image types such as JPEG, PNG and TIFF you need to add appropiate libraries which Nano-X will call. You will probably have already downloaded these libraries as described in section one.

The BMP images in the c:\djgpp\microwin\src\mwin\bmp directory are converted by executing the Makefile using the convbmp.c program to an internal Nano-X image format and then compiled into the libimages.a library to be used by the demo programs. The malpha demo e.g. uses the car8.bmp image this way.

You can also use the GrDrawImageFromFile() function to read an image from a file and display it in a window. The GrLoadImageFromFile() function on the other hand will load the image from file and return an image id number. This allows this image to be displayed later in the program. The nxview demo uses this function.

Unfortunately I could only get BMP and PPM images to work. The devimage.c code needs to be adapted for DJGPP. Therefore support for GIF, JPEG, PNG and TIFF images is not enabled as standard in the makefile for DJGPP. Just uncomment the lines provided for these image formats if you want to test this.

<u>6. Fonts</u>

Nano-X supports "bdf" bitmap fonts, Windows "fnt" bitmap fonts, X11 "pcf" fonts, Adobe Type1 or postscript fonts and "ttf" truetype fonts. Plus japanese and chinese fonts.

The "pcf" fonts can be stored using z compression and will have a "gz" ending then. Nano-X then needs the "libz.a" library and its header files to read these. The "fnt" fonts could also be stored as gz compressed files.

Nano-X is compiled with two compiled-in system fonts. This are "winFreeSansSerif11x13" as variable font and "X6x13" as fixed font in the fonts directory. The demos usually use these fonts.

You can also extend these compiled-in fonts. For this you have to modify the "drivers/genfont.c" file. Add the desired fonts to the "gen_fonts" array there and modify NUMBER_FONTS number in "include/genfont.h" accordingly. Then change the Makefile in the fonts directory to include that font. For a file to be compiled-in you have to use a bdf font file and the convbdf program to convert that into a C program file. Then you compile that file and may add it to a library. The Makefile in the fonts directory will do this for several fonts.

The pcfdemo.c in demos/nanox allows to display a PCF font. The pcf font file can also be stored in a GZ archive since it takes quite some space on disk. If you specify such a file for pcfdemo on the command line, Nano-X will use the libz.a library to decompress this font file before using it. Furthermore this demo program allows to display a FNT plus a TrueType font in spite of its name too.

Demo2.c can also be compiled with various other fonts. If demo2.exe is in the bin directory add e.g. "../fonts/fnt" before the name of the font here.

Type 1 support is made with the library "libt1.a".

For TrueType support Nano-X needs the external library "libfreetype.a" plus a long list of header files. Set the environment variable as described above in section two for TrueType support. The font directory cannot be set in the makefile here successfully.

You also need to modify the Makefile in the engines directory to include font_freetype2.o in the objects to compile. Just uncomment that line provided.

I downloaded mupdf for DOS which comes with a compiled "libfreetype.a" library.
This is the link: http://www.ulozto.cz/7940392/mupdf-07-djgpp-rar . You have to guess Czech words to download it from there though. The libfreetype.a library contains debug symbols. You can use "strip −−strip-debug libfreetype.a" to remove those and get a much smaller file size. Provided the strip.exe program is in your DJGPP bin directory.

The fontdemo.c program will show the different type sizes of a TrueType font on the screen. The environment variable set TTFONTDIR which has been set with the batch file in section 2 will determine where the fontdemo.c program will look for the font DejaVuSans.ttf.

<u>7. Developing programs using Nano-X</u>

To develop your own programs using Nano-X you could take the following steps:

1. create directory for your files e.g. c:myprogs.
2. copy libmwin.a, libnano-x and libimages.a from "src\lib" into c:\djgpp\lib.
3. make a subdirectory called "mwin" in c:\djgpp\include and put the files from "src\include" into that.
4. copy mtest.c from "src\demos\mwin" into c:myprogs since we use that as example.
5. run start.bat in c:\djgpp as described in section one to set the path and environment.
6. edit mtest.c and change the include line to: "#include <mwin\windows.h>".
7. compile with: "gcc -Wall -g -s -O3 -o mtest.exe mtest.c -lmwin -lfreetype -lz"
8. run the mtest.exe program.

For other programs you may have to include fonts or other libraries too. Step 7 could be put into a batch file.

The resulting programs have a large size on disk compared with other DOS programs. I compiled with the "-s" switch which reduced the size by over 70%. This port has not worked further on reducing the size. Here are some recommendations to do that, e.g. remove the functions to read command line options:

http://www.delorie.com/djgpp/v2faq/faq8_14.html

Also the fonts and images could be compiled as DXE files, DJGPP's way to make a dynamic loadable library: http://www.delorie.com/djgpp/v2faq/faq22_15.html

Some additional remarks regarding program development:

If you need GUI widgets like buttons etc. you can use the Microwindows API which includes these just like MS-Windows. The Nano-X API just has the functionality of a graphics library. There were NXWidgets in earlier versions but these are not longer included since FLTK, wxWidgets or GTK can be used with Nano-X/NXLIB when used with a Linux operating system.

Nano-X uses DPRINTF(), EPRINTF() and FPRINTF() for output of error messages. These are not properly displayed when the screen is in TrueColor mode. You can use the DJGPP redir program to redirect standard error and standard output to files. E.g.
"redir -e err.log -o text.log nxview tux.gif" will write standard error messages into err.log and messages for standard out (the default screen in text mode) to text.log. You can use redir also with the make statement. The warnings will then be in err.log and the executed statements in text.log.
For debugging you can insert "printf()" statements in the code. You can either use redir as described above or run e.g. "mtest.exe >test.log" and the ouput of these printf() statements will be written into the file test.log.

Some demos will not return to text mode if they terminate due to an error. The DOS command line will then result in colored pixels on the screen. I wrote the "vcls.exe" utitlity which can be called then to set the screen back to text mode.

Read the following page about handling of slashes "/" by DJGPP and converting Linux device names to DOS: http://www.delorie.com/djgpp/doc/libc/libc_626.html

This page also mentions that getting a program to compile with DJGPP is usually not enough, you also have to check if it works as it does under Linux. Then you have to apply the necessary changes. I did not do that completely when porting Nano-X to DOS.

If you are using the microwindows interface, there is a makedlg.c program in the "mwin\src\contrib\makedlg" directory which will make a dialog for your program using a resource file.

8. Compiling NXLIB

If you have a package which uses X11 on Linux and you want to use that with Nano-X you can adapt the code of this package to work with Nano-X.

Or you use NXLIB which is an X11 conversion library for Nano-X. This provides a full X11 interface to application programs so these do not require code changes to work with Nano-X. The NXLIB library converts X11 calls to nano-X calls. Functions which are not implemented just return 0. This is done in the stub.c program, where you can add functions that may be called by your application and which are not implemented yet.

With DJGPP you will compile a static library of NXLIB and then compile the Linux application including this library. However, you will probably also need the GNU utilities converted to DJGPP as recommended for download in section one. Perl is required to compile NXLIB.

So download NXLIB from the Microwindows website and use 7zip to unpack it into the myprogs directory. I used nxlib-src-snapshot.tar.gz of 29th July 2011 which supports fonts with FLTK 1.3.0 unlike earlier versions.

I describe here how to make the required changes by hand instead of using a patch file. I think this will be easier to apply to new versions than a patch.

If you need international keyboard support you should replace the existing StrKeysym.c and NextEvent.c files in the NXLIB directory with the ones supplied with this document.

To compile NXLIB with DJGPP you have to set the directories in the makefile along the lines of this example, which assumes that you have made the steps as described in section 7 before:

```
MWIN_INCLUDE=/djgpp/include/mwin -I/myprogs/nxlib
MWIN_LIB=/djgpp/lib

X11_INCLUDE=/usr/include

X11_RGBTXT=fonts/rgb.txt

INSTALL_DIR=/myprogs/nxlib/lib
SHAREDLIB=N
INCLUDE_XRM=N
```

To suppress debug messages you should comment out line 14 as shown below. These go to stout and will corrupt the display if not redirected when calling the application program:
#CFLAGS += -DDEBUG=1 -g

Also you may add the "-s" flag in line 70 to reduce the binary size of the library by 75%. The line then looks like that:
CFLAGS += -Wall -s

If you want to run through the code with GDB you should add the -g flag. This is set if GDB=1 is set in the makefile:
CFLAGS = $(INC) -Wall -g

Then remove "./" in front of keysymstr.h in the makefile:

```
keysymstr.h:
      perl ./keymap.pl $(X11_INCLUDE)/X11 > keysymstr.h
```

Then make a directory called "lib" in "/myprogs/nxlib/"

I recommend to compile NXLIB with the original X11 header files. So copy the directory "usr/include/X11" from a Linux installation and put this as "usr/include/X11" on the disk where you compile NXLIB.

You will also need the files "keyboard.h" and "kb.h" which you can copy from any linux distribution from the directory "usr/lib/linux". Make a directory "/usr/include/linux" and put the files into that. Then:

In kd.h you have to comment out the line:
```
//#include <linux/types.h>
```
and in keyboard.h the line:
```
//#include <linux/wait.h>
```

Then patch the "StrKeysym.c" file by using // to comment out:
```
//#if linux
```
and the corresponding
```
//#endif /* linux*/
```

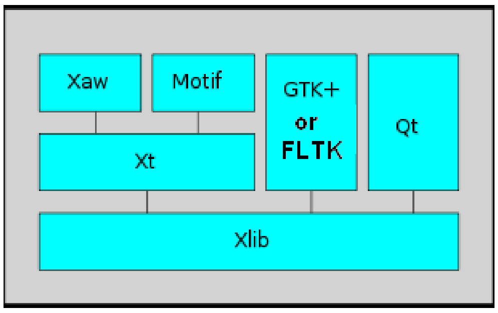
Now you can enter "make" and a new libNX11.a will be generated in the nxlib directory. Make sure djgpp is in your path. You may use "strip --strip-debug libXN11.a" to reduce the size.

I do not recommend to use "make distclean" here as mentioned in the makefile. However, if you have done that and you have to repeat make, a Perl script will generate the keysymstr.h file needed for the compilation and this is removed again by "make distclean".

Now that we have successfully made NXLIB we can copy libNX11.a into /djgpp/lib and "usr/include/X11" into /djgpp/include/X11.

9. Compiling a program using NXLIB

There are only a few application programs which are based on X directly. Usually GTK, FLTK, Enlightenment etc. are used to have GUI widgets. In the next section we will see how to compile FLTK with NXLIB. Below is an image how common libraries are based on Xlib. NXlib is compatible to just Xlib. If a program uses Xt, Xaw or Motif, you will not be able to port this to DOS. You would have to port Xt or Xaw to DOS yourself.



You can see that a program uses Xt when it includes "X11/Intrinsic.h". When it includes "X11/Xaw/*.h" it uses the Athena widgets.

For a test of our newly compiled NXLIB we will compile a demo program which depends on X. Here is a very simple X program which I call xhello.c. Compile this with:

```
gcc -I/djgpp/include/mwin -o xhello.exe xhello.c -lNX11 -lnano-X -lz -lfreetype
```

If you have compiled nano-X without z and freetype support, you probably will not need these libraries on the command line.

Please keep the order of the -l statements here since the libraries depend on each other.

The code for xhello.c is included in the directory where you found this document. The example will also handle keyboard input and display that on the screen.

10. Compiling FLTK with NXLIB and Nano-X

It is possible to compile FLTK with NXLIB and DJGPP. I suggest to download the FLTK 1.3.0 package from the FLTK website to get the latest version of this branch. Use 7zip to unpack it into the myprogs directory. Rename the directory name to fltk130 to avoid the dots in the directory name.

To compile FLTK you need the full set of GNU utilities installed - bash, autoconf, automake, sed, textutils, sh-utils, perl, grep, awk, fileutils, findutils, diffutils, m4 and above all the GNU c++ compiler gpp. Some of these packages try to install the same file again, make sure to keep the latest version of it.

FLTK is written in C++ and the source files have the extension cxx. This is just a convention, mostly the extension cpp is used for C++ source code file. We will continue to use cxx here in this section.

Test if gpp really works by compiling this simple c++ program:

```
#include <iostream>
using namespace std;
int main()
{
     std::cout <<"Hello World" << std::endl;
}
```

Save this program as world.cxx and compile it with this line:

```
        gpp -o world.exe world.cxx
```

If you can run world.exe you have C++ properly installed.

Check that libNX11.a and libnano-X.a are in the "djgpp/lib" directory and the X11 header files are in the "djgpp/include/X11" directory. Here it is assumed that you want to use the freetype library with FLTK and thus the instructions following imply that.

Now make the following changes to the files in the fltk130 directory:

If you want to try to enable threads (not recomended) change line 792 to:

```
        for flag in -lgthreads -lpthread -gthreads; do
```

Then open the configure file and comment out the lines 11555-11559 with "#" since we know we have X11 installed - at least sort of. Then change the line 11566 to:

```
        LIBS="$LIBS -lNX11 -lnano-X -lfreetype $X_EXTRA_LIBS"
```

Set -lNX11 before -lnano-X here.

If you want to try to enable threads (not recomended) change line 9791 to:

```
                for flag in -lpthread -lpthread -pthread; do
```

Also add - lwatt in lines 9795 und above in line 11566 if you want to try pthread.

Then edit the file fl_open_uri.cxx in the src directory and comment out line 286:

```
// sigaddset(&set, SIGCHLD);
```

DJGPP does not support SIGCHLD.

Finally the sudoku.cxx example will not compile. So go into the "fltk130\test" directory and open the Makefile. Remove sudoku from CPPFILES =\ and ALL = \.

Make sure you have got truetype fonts available in the \usr\share\fonts\truetype\ttf-dejavu directory or you will get no text with FLTK. Above all put the file "fonts.alias" into this directory to make it work. This file is included in the directory where you found this document and should be copied together with the supplied fonts.

I applied also a patch from Dmitij called "Grad1" which adds another scheme to FLTK. That results in buttons etc. which look more in Win7 style.

Now you can enter "bash" at the command line - if you are in the fltk130 directory. Then enter:

```
 "./configure --disable-gl --disable-xinerama --disable-xdbe --disable-threads".
```

After configure has terminated you can enter "make". This will compile the FLTK libraries, the FLUID application, the examples in the test directory and and then start to compile the documentation. Since I did not have man installed this did not work but I did not bother to get that working.

In the fltk130/lib directory there will be three new libraries: libfltk.a, libfltk_forms.a and libfltk_images.a. Copy these into "djgpp/lib". Also copy the FL directory into "djgpp/include/" so you get "djgpp/include/FL".

There are two examples in the test directory which need a few changes to run properly. The first is native-filechooser.cxx. Here you have to change "." to "" in line 87. The next is file_chooser.cxx where the same has to be applied in line 104 plus the lines 124-127 have to be replaced with "filter->value("");
Both look for the HOME environment variable. So e.g. enter "set HOME=/myprogs" in the command line. You can then select that in the "favorites" drop box.

You should run the editor example in the bash shell to handle the \ and / problem or edit the source code at the Win32 parts.

I did not get the threads example to work and therefore I recommend to run ./configure with the −−disable-threads switch.

FLTK allows to select different schemes using the "FLTK_SCHEME" environment variable. These look better than the default scheme. The grad1 scheme is only available if Dmitrij's patch is applied. E.g.:

```
set FLTK_SCHEME=plastic
set FLTK_SCHEME=gtk+
set FLTK_SCHEME=grad1
```

11. Compile and run FLTK programs

Let's start with the most simple FLTK program: this will create a FLTK window and draw a box in it with the message "Hello World". This is the hello.cxx example in the test directory. It has already been compiled with make but lets see how we can compile this again in a separate directory. So copy hello.cxx to fltkhello.cxx in \myprogs.

You can compile it with the following command:

```
"gpp -I/djgpp/include -o fltkhello fltkhello.cxx -L/djgpp/lib -lfltk -lNX11 -lnano-X -lfreetype"
```

After it is compiled, run this program with the following command, since it will write some messages to stderr: "redir -e err.log -o text.log fltkhello.exe" . If you have compiled NXlib without the debug switch these message will not be generated. However you can only move the mouse if you redirect stdout. The screen driver will do that by default.

Exit the fltkhello program using the ESC key. FLTK will not call XCloseDisplay() when exiting and therefore the VESA mode will not be set back to text mode. There are two ways to overcome this:

a) if you write a new application add the function
`fl_close_display();`
just before your
`exit(0);`

to be able to use this function you have to include this header file:
`#include <FL/x.H>`

b) or call my utility vcls.exe (for vesa cls) after terminating the fltk program. E.g. make the following two-line batch file to run the samples in the test directory:

```
%1.exe
vcls.exe
```

Now just enter the program name after the batch file name to run it.

This is the code for the vcls utility:

```
/* Compile with: gcc -s -o vcls.exe vcls.c */
#include <dos.h>
int main(void)
{
   /* return to text mode */
   union REGS regs;
   regs.h.ah=0x00;
   regs.h.al=0x03;
   int86(0x10, &regs, &regs);
   return(0);
}
```

If you have run the FLTK program without this batch file, you have to enter "vcls" blindfolded to return to text mode again.

12. Debugging programs with GDB

A problem with interactive debugging programs based on Nano-X in DOS is the fact that the display is set to SVGA mode while debug messages e.g. from GDB are displayed in text mode. Therefore these cannot be seen while the screen is in SVGA mode.

Therefore I implemented the key combination ALT-comma. If you press ALT-comma, the screen will be set in text mode and all output for the SVGA screen will remain in the internal screen buffer.
If you press ALT-comma again, the screen will be set into SVGA mode again and the display restored from the internal buffer. This allows to use GDB as an interactive debugger.

Let's make a simple GDB session using our xhello.c example mentioned in section 9.

To add debugging information for GDB we have to compile the example with the -g flag and without the -s flag:
gcc -g -I/djgpp/include/mwin -o xhello.exe xhello.c -lNX11 -lnano-X -lz -lfreetype

Then we load it into GDB by entering: "gdb xhello" The (gdb) prompt will appear.

If we enter "q" now, we will exit GDB again.

Entering "l 1" (list from line 1) will display the first 10 lines of our xhello example. Pressing enter will display the next lines. Entering "l 50" will display ten lines with line 50 in the middle.

We can also specify a function name here: "l terminate" will display our "terminate" function.
If there are several source files linked together, you have to preceed the line number with the name of that file and a colon. So you can e.g. enter "l foo.c:10" to list line 10 and the following ones in source file "foo.c".

Before you run the program, you usually set some breakpoints first. We will set a breakpoint if one clicks into the window. This is handled in "case ButtonPress:". Since the code below that is commented out, we set the breakpoint on the "break" statement in line 98.

Entering "b 98" sets a breakpoint at statement 98. "clear 98" would remove that breakpoint again.
We can also enter a function name here, so "b terminate" will break when this function is called.
You can also add a condition to the breakpoint. If you enter "b 98 blackColor = 0" the breakpoint will only be triggered if blackColor is zero.

Now we start our the program by entering "r" at the (gdb) prompt.

I have to say that a problem here is that when the breakpoint is reached, GDB will output text while the screen is still in SVGA mode.
So we have to proceed this way: move the mouse into the window without clicking the button yet and press ALT-comma to set the screen into text mode. Then click the mouse button.
GDB will report that the breakpoint has been reached and display its (gdb) prompt which allows you to enter further commands.

If you forgot to do that, GDB will have interrupted your program and send its (gdb) prompt to the SVGA screen. What you can do is enter "c"plus enter for continue blindfolded and try again.

To inspect the value of a variable you can use the "p" command. So "p text" will display "Hello world" while "p whiteColor" (capital C!) will display the numeric value of that color. Local variables within functions can only be displayed while you are within that function.

To see the color value in hex you can enter: "p /x whiteColor". These format specifiers are familiar from the printf statement in C. In fact you can also enter statements like that: "printf "Colorvalue: %d",whiteColor". Do not use parentheses here.

If you enter "display whiteColor" this variable will be displayed automatically whenever the program is interrupted. "undisplay" will remove that again, but you have to specify the number for this command. "info display" will output the numbers for the display commands specified so far.

You can also change variables with GDB. If you enter "set blackColor = 10000" the updated part of the window background will be green.

If you enter "c" the program will continue. We can also interrupt it and return to GDB. For this enter ALT-comma and then CTRL-C. Usually you need several attempts till CTRL-C is accepted. Then the (gdb) prompt will appear and you can enter commands again.

You could now search for an expression: "search dpy" will display the next line containing "dpy" while "reverse-search Hello" will move up the code to "Hello World".

Here you could enter "s" to single-step through the code. "s 10" will make ten steps. If you do not want to step into a function, you can step over that by entering "n" for "next" to step to the line following that function call.

Now enter "c" to make the program continue again and click into the window after pressing ALT-comma to trigger our breakpoint.

Using the "call" statement we can call functions from the (gdb) prompt. If we enter "call terminate(dpy)" now, the program will exit.

However, if we enter "call terminate(1)" instead we will get an error. Now we can use the "bt" or backtrace command will allow to display the lines of code before the error occured.

If you want to trace into Nano-X or NXlib you have to compile these with the -g switch too. There is a GDB flag for that in the src\Makefile.

Commands covered in this walk-through:

q - quit
r - run program
c - continue program
s - single step through program
n - step through program, skipping current function
CTRL-C interrupt program and return to GDB
b - set breakpoint
call - call function
bt - backtrace

p - display value of variable
display - set variable to display on each breakpoint
set - change value of variable

l - list lines of code
search - search expression in code

A manual for GDB can be found here:
http://sourceware.org/gdb/onlinedocs/gdb/index.html#Top
As an alternative, the IDE Rhide can be used with its integrated debugging support.

September 2011 Georg Potthast - mailbox@georgpotthast.de